Blitz-64: Memory-Mapped I/O Devices

Harry H. Porter III

HHPorter3@gmail.com

14 December 2023

With the Blitz-64 computer architecture, external devices (i.e., peripherals) are memory-mapped into the main memory address space. This document describes how the processor core accesses and controls I/O devices by loading from and storing into memory. Various Blitz-64 implementations will support different devices. This document specifies the devices supported by the Blitz-64 emulator and by the MicroBlitz Verilog/FPGA implementation.

Table of Contents

Chapter 1: Memory-Mapped I/O 4

Quick Summary 4
Overview 4

Chapter 2: Platform-Level Interrupt Controller (PLIC) 9

Quick Summary 9
PLIC: Platform-Level Interrupt Controller 9
This Specification 11
Memory-Mapped PLIC Registers 12
Enabled Devices 13
Claiming Interrupts 13
Retiring Interrupts 13
From the Trap Handler's Perspective 13
Level-Triggered and Edge-Triggered Interrupts 16
Level-Triggered 17
Edge-Triggered 18
Claiming Multiple Interrupts Not Allowed 19
Discussion of Handler Approaches 20
Implementation in the Emulator 22
Interrupt Priorities 25

Chapter 3: UART0 26

Quick Summary 26
Introduction 26
Memory-Mapped I/O Registers 27
Operation 28
Notes 29

Chapter 4: DISK0 31

Quick Summary 31
Introduction 31
Memory-Mapped I/O Registers 32
Operation 33
Emulator Implementation 36
Notes 37

Chapter 5: DMA Controller 38

Quick Summary 38
Background 38
The Blitz DMA Controller 39
Moving Blocks of Memory 42
Wait for Task to Complete 42
Zeroing Blocks of Memory 43
SHA-256 44
AES-256 45

Chapter 6: Other Ideas 50

Quick Summary 50
Overview 50
Lock Controller 50
Digital I/O Pins and LEDs 54
SPI / MicroSD Card Slot 55
Adjacent Core Links 56
HDMI, USB, WiFi, etc. 58

Chapter 7: The CONTROL and 59

CONTROLU Instructions 59

Quick Summary 59
CONTROL and CONTROLU 59
In the Emulator and Verilog/FPGA 60
Digital Read / Write 61
Halt 62
Serial / UART Control 62
ENABLE_KERNEL 63
SET_STATUS 63
TLB_DEBUG 63

Chapter 1: Memory-Mapped I/O

Quick Summary

- Each I/O device is allocated one or more pages.
- The memory-mapped I/O pages are located in a dedicated region of addresses.
- The memory-mapped I/O region is 16 GiBytes (1 Mi Pages).
- The memory-mapped I/O region begins at address 0x4_0000_0000.
- Code running in kernel mode has full access to the memory-mapped I/O region.
- The I/O pages may optionally be mapped into virtual address spaces.
- The Boot ROM Area is treated as a memory-mapped I/O region.

Overview

The Blitz-64 architecture does not contain instructions that are dedicated to input or output.

Instead, all I/O devices are **memory-mapped**, which means they are accessed using LOAD and STORE instructions. In addition, instructions can also be FETCHed from memory-mapped I/O regions. For example, instructions are fetched from the Boot ROM Area.

Each device is assigned to, and located within, one or more pages. In other words, the starting address for a device's address range will be page-aligned and the amount of address space the device consumes will be a multiple of the page size, which is 16 KiBytes.

The following address range is set aside for memory-mapped I/O pages:

		Size	
Starting Addr	**Ending Addr**	**Bytes**	**Pages**
4_0000_0000	7_FFFF_FFFF	16 GiBytes	16,777,216

In the layout of the memory-mapped I/O region, the various I/O devices will be ordered and laid out sequentially, one after the other. They will not overlap and different devices will be on different pages.[1]

The exact layout of the memory-mapped I/O regions is implementation-dependent.

Allocating the memory-mapped I/O address space in units of pages is mandated for the following reason: it allows the kernel to use address translation to map the pages into various virtual address spaces. At runtime, the Memory Management Unit will use page tables to map a LOAD or STORE from a virtual address to the physical address of the device. Thus, the kernel can use the paging mechanism to make an individual memory-mapped I/O device available to one address space, but hidden and invisible to all other address spaces.

In most cases, the device driver for a particular device will run as a user-mode program. The pages for the device being managed are mapped into the address space of the driver program. This approach frees the kernel from the overhead of dealing with many devices. More importantly, it allows device drivers to be dynamically loaded, started, and stopped in a safe fashion. If a device driver is buggy or contains malicious code, the damage is limited to the device in question; it cannot modify other devices or corrupt kernel memory. Moving most device drivers out of the kernel is critical for security, as well as flexibility.

Nevertheless, a few devices will undoubtedly be managed directly by the kernel. The pages for such a device would not be mapped into any virtual address space and the kernel code would address the pages directly.

Like normal memory pages, I/O pages that are mapped into virtual spaces may have any combination of permissions.

By not mapping a memory-mapped I/O page into a virtual address space, the kernel prevents user mode code from accessing the device. If a page is mapped, then it will be readable. In addition, the kernel may mark the page as writable and/or executable. Normally, the pages for I/O devices would be marked writable (allowing the code to update/alter/command the device) but not executable.

---

[1] It is allowed for implementation to place unused gaps between the regions, if this is convenient. If there is an expectation that a region will grow in subsequent implementations or that some devices may be implemented optionally in different versions, then those pages should be pre-allocated, set aside, and documented as such.

The Blitz-64 specification does not fully specify the nature of all I/O devices available on a given system. In fact, different implementations will have different devices. In other words, which devices are present and how they function will vary between implementations.

Each implementation must specify:

- Which I/O devices are present
- Where each device is located in the physical address space
- How many pages are allocated to each device
- Exactly how the device functions and how it is used

Below is an example placement of memory-mapped I/O devices. (This happens to be the default memory map for the devices implemented by the Blitz-64 emulator.)

		Size		
Device	**Starting Addr**	**Hex**	**Bytes**	**Pages**
Boot ROM Area	4_0000_0000	10_0000	1 MiBytes	64
Secure Storage Area	4_0010_0000	10_0000	1 MiBytes	64
PLIC	4_0020_0000	4000	16 KiBytes	1
UART	4_0020_4000	4000	16 KiBytes	1
DISK	4_0020_8000	4000	16 KiBytes	1
DMA Device	4_0020_c000	4000	16 KiBytes	1
Host Device	4_0021_0000	4000	16 KiBytes	1

The **Boot ROM Area** and the **Secure Storage Area** are documented in the *"Blitz-64: Instruction Set Architecture Reference Manual"*.

Subsequent chapters here define, describe, and document the other devices. Of course, different implementations may choose different specifications for various memory-mapped peripheral I/O devices. In this document, we describe what is implemented by the Blitz-64 emulator.

Another chapter sketches out ideas for other hypothetical devices, including:

- Lock Controller
- Digital I/O Pins
- SPI / MicroSD Card
- Adjacent Core Links

- HDMI, USB, WiFi

Finally, one chapter includes the specification for the **CONTROL** and **CONTROLU** instructions as they are implemented by the emulator.

Chapter 2: Platform-Level Interrupt Controller (PLIC)

Quick Summary

- There is one **PLIC** for each multicore system.
- All I/O devices, including the PLIC itself, are **memory-mapped**.
- Devices send interrupt requests to the PLIC.
- The PLIC will forward the interrupt to the cores.
  - — One core will **claim** the interrupt.
  - — This core will service the interrupt (via a **handler routine**).
  - — When complete, the handler will **retire** the interrupt.
- Only one interrupt per device may be actively claimed at any time.
  - — Multiple interrupts from one device are handled sequentially.
  - — The PLIC enforces this sequential linearization of interrupt handling.
- During set-up, the PLIC is configured with an **ENABLE_ARRAY**.
  - — Device D may (or may not) be enabled for Core C.
  - — When device D interrupts, only the enabled cores will be interrupted.
  - — The PLIC allows only one enabled core to successfully claim the interrupt.

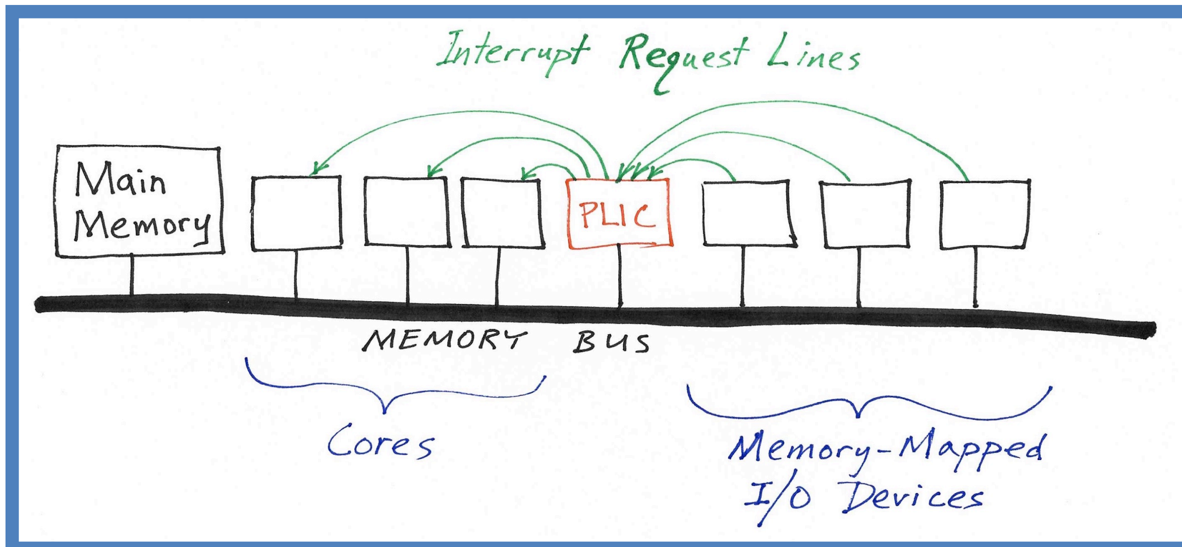
PLIC: Platform-Level Interrupt Controller

From time-to-time I/O devices will generate interrupts and each interrupt must be routed to a core to be handled. A multicore Blitz System will contain a single PLIC, which will route interrupts from devices to the cores.

Of course, there will be some interrupt sources that are local to a particular core. For example, each core will have its own timer and each timer interrupt will go to only that core. For such devices, the interrupt will not go through the PLIC. Instead, there are specific interrupt types (such as “Timer Interrupt”) and the device will interrupt the core directly.

Nevertheless, many devices will be shared among the cores. Each interrupt generated by a such a shared device goes through the PLIC, which will then interrupt one or more cores. In this discussion of the PLIC, we ignore local devices, since the PLIC is only used for shared devices.

Figure: Basic PLIC Architecture



Typically, all cores and the PLIC are located on the same IC chip. The interrupting device may be located on the same IC chip as the cores or it may be located elsewhere. When a device raises an interrupt, the PLIC is notified via a direct connection, which would normally be a single wire.

The PLIC will then cause a “PLIC Interrupt” to be raised in one or more cores. All interrupted cores will execute the trap handler code. Of course, if a core has interrupts disabled, there will be a delay before interrupts are re-enabled, the PLIC interrupt is serviced, and the handler code begins.

Only one core will “claim” the interrupt. This core will then (presumably) execute actions to service the I/O device. All other cores will find that the interrupt has been claimed by another core and will exit their trap handling and resume execution of the interrupted code.

The first step of the trap handler is to contact the PLIC to “claim” the interrupt. The PLIC will then indicate whether the claim operation is successful or not. If the claim operation is successful, the core will go on to deal with the I/O device. If the claim operation is not successful, the core will return to other tasks.

Every I/O device is memory-mapped, and each will occupy its own distinct region of the physical address space. To access a device, a core will use LOAD and STORE instructions. More precisely, to send data to a device, the core will STORE to specific locations. To retrieve status and data from a device, the core will use LOAD instructions. Such memory locations are often called "I/O registers", although they should not be confused with registers within the core.

To determine the addresses of the various I/O registers associated with a particular device, consult the documentation associated with that device.

The PLIC is also treated as a memory-mapped device. There are a number of I/O registers associated with the PLIC, and these will be described in this document. For example, to set-up and initialize the PLIC before operation, a core will STORE into specific PLIC registers. Likewise, to claim an interrupt and the complete the interrupt, a core will LOAD and STORE into addresses mapped to the PLIC.

In addition to the memory-mapped I/O registers, there are separate wires associated with the PLIC. Each device has a single wire to the PLIC, which is used by the device to signal an interrupt. Also, there is a wire from the PLIC to each core, which is used by the PLIC to signal a PLIC Interrupt to that core.

This Specification

The PLIC design and specification presented here is similar to, but distinct from the Risc-V PLIC. Furthermore, this design may be changed in the future. Instead of version numbers, we use the dates on this document and the implementation itself.

This design supports…

Maximum Number of Cores	128
Maximum Number of Devices	64

The cores are numbered 0, 1, 2, … 127.

The devices are numbered 0, 1, 2, … 63.

There is no support for interrupt "priorities". All devices are treated as equally important.[2]

Memory-Mapped PLIC Registers

EDGE_TRIGGERED_ARRAY — 64 bits, one bit per device
    0 = This device is level triggered
    1 = This device is edge triggered
ENABLE_ARRAY — 128 × 64 bits, one doubleword per core
    1 = This device can interrupt this core
    0 = This core will not get interrupts from this device
CLAIM_ARRAY — 128 doublewords, 1 per core
    Read/LOAD to determine which device has interrupted.
    Write/STORE to retire the current interrupt

These registers are memory-mapped. Memory-mapped registers can be read and written by the cores.

The PLIC would typically be memory-mapped to the page with address **0x4_0020_0000**.[3] The offsets of these registers are:

Offset (decimal)	Typical Addr	SizeInBytes	Register
0x0000 (0)	4_0020_0000	0x0008 (8)	EDGE_TRIGGERED_ARRAY
0x0008 (8)	4_0020_0008	0x0400 (1024)	ENABLE_ARRAY
0x0408 (1032)	4_0020_0408	0x0400 (1024)	CLAIM_ARRAY
0x0808 (2056)	*4_0020_0808*		

Normally, the **EDGE_TRIGGERED_ARRAY** and the **ENABLE_ARRAY** are written during setup and initialization and would not be changed thereafter.

[2] Modern devices are typically managed by their own microcontrollers that provide data buffering, eliminating the need for super-fast interrupt handling. Furthermore, in a multi-core system where any core can service any device, there is usually a core available whenever any interrupt occurs.

[3] With the Blitz-64 emulator, this address is one of the **emulation parameters**, and can be adjusted.

Enabled Devices

When a device raises an interrupt, which cores should be notified? This is determined by how the **ENABLED_ARRAY** was initialized on start-up. This array determines which cores will be notified when a particular device requests an interrupt. It also determines, for a given core, which devices are able to trigger an interrupt on that core.

Claiming Interrupts

Whenever a core receives a "PLIC Interrupt", it should LOAD from the element of the **CLAIM_ARRAY** corresponding to that core. If the value retrieved from the PLIC is -1, it means the interrupt has been previously claimed by another core or that the interrupt is no longer pending.[4] Otherwise, the value retrieved from the register will be 0…63 to indicate which device is raising an interrupt.

Retiring Interrupts

A core that has successfully claimed an interrupt from device X should retire that interrupt after handling it. This is done by STORING into the core's claim word. The exact value stored does not matter and is ignored by the PLIC.

From the Trap Handler's Perspective

When a device has generated (i.e., "signaled" or "raised") an interrupt, the PLIC will generate a "PLIC Interrupt" on all cores for which that device is enabled. The PLIC determines this from the **ENABLE_ARRAY**, which tells for each core, which devices are allowed to cause interrupts on that core. Interrupts from devices that are not enabled are never forwarded to that core.

Focussing on one core, if interrupts are disabled on that core (i.e., if the "Interrupts Enabled" bit in **csr_status** is 0), that core will continue executing normally. On the

[4] How can the interrupt "no longer be pending"? Perhaps the interrupt has been claimed by another core, the handler has run to completion on that core, and the interrupt has been retired, all before some other core has even attempted to claim the interrupt.

other hand, if interrupts are enabled (or once interrupts are re-enabled), trap processing will begin and a jump to the global trap handler will occur. At the time of the trap, the **csr_cause** will be set to the code for a "PLIC Interrupt".

If interrupts are disabled at core X and the interrupt is successfully claimed by some other core Y before interrupts are re-enabled on X, the interrupt may or may not disappear. In other words, the PLIC Interrupt at core X may remain pending until interrupts are re-enabled or it may disappear. If the interrupt disappears, then trap processing never occurs and, from the perspective of core X, it is as if the PLIC Interrupt never occurred. This is an implementation detail to be decided by the PLIC implementor.[5]

Once the trap handling begins for the PLIC interrupt, the interrupt must first be "claimed". Since several cores may be notified and begin trap processing more or less simultaneously, and since only one must service the interrupt, all interrupted cores must communicate with the PLIC. The PLIC will give the interrupt to exactly one core. That is, the claim operation will be successful on one core and will fail on all other cores attempting to claim that same interrupt.

Furthermore, the core could have several devices enabled, so after the PLIC Interrupt, the handler must determine which device is requesting service, and the core gets this information when it LOADs from the **CLAIM_ARRAY**.

There is a "claim register" for each of the cores. Each claim register is a doubleword (i.e., 64 bits) which will contain an integer. To claim an interrupt, a core will LOAD from its register in the **CLAIM_ARRAY**. For example, core 5 will LOAD from **CLAIM_ARRAY[5]**.

The PLIC will determine whether or not a claim operation is successful for any core trying to claim it, giving the interrupt to exactly one core, and not to any other cores which are also trying to claim the interrupt. If the operation is successful, the PLIC will return the number of the device that is interrupting. If the claim operation fails, the PLIC will return -1 to the core.

[5] Furthermore, it may be that such an interrupt sometimes remains pending and sometimes disappears, at the discretion of the PLIC.

If the claim operation fails, then the trap handler will return to the interrupted code and will do other things.[6] But if the claim is successful, the trap handler will…

- Determine which device is interrupting
- Execute code specific to that device
- Communicate directly with that device as needed to process the interrupt
- Retire the interrupt by communicating to the PLIC
- Return to the interrupted code

The value returned from LOADing the **CLAIM_ARRAY** will indicate which device is interrupting and can be used to dispatch to code specific to that device. After the interrupt is handled, it must be retired by notifying the PLIC that the core has completed its handling the interrupt.

To "retire" the interrupt, the core will STORE into the **CLAIM_ARRAY** register for that core in the PLIC. As mentioned, the **CLAIM_ARRAY** register has one doubleword for each device. The core may store any value into this doubleword; the actual value is ignored by the PLIC.

Once an interrupt from some device has been claimed by one core, no further interrupts from that device will be dispatched by the PLIC until that interrupt is retired. In other words, if an interrupt from some device (say X) has been claimed but not yet retired, then subsequent interrupts from device X will not cause a "PLIC Interrupt" in any core, nor will any attempts to claim an interrupt be successful.

In order to use the PLIC correctly, the following rules must be respected by the cores:

- A core must LOAD from and STORE to only the **CLAIM_ARRAY** register associated with that core. For example, core 5 should never mess with the register associated with core 9.

- A core must not execute a "retire" operation unless it has first successfully "claimed" an interrupt. In other words, each retire operation should be preceded by a successful claim.

[6] Presumably. Here, we describe the normal operation of the core and assume all handlers behave correctly.

- A core should not try to claim an interrupt unless it has retired the previous interrupt and the PLIC has subsequently generated a "PLIC Interrupt".
- The **ENABLE_ARRAY** and the **EDGE_TRIGGERED_ARRAY** should only be modified before the first interrupt occurs.

Violations of these rules may confuse the PLIC and cause erroneous operation.

Level-Triggered and Edge-Triggered Interrupts

There will be a single line from each device to the PLIC and the device will use that wire to signal (i.e., request/raise) an interrupt[7]. The interrupt request line (from the device to the PLIC) can be configured as either:

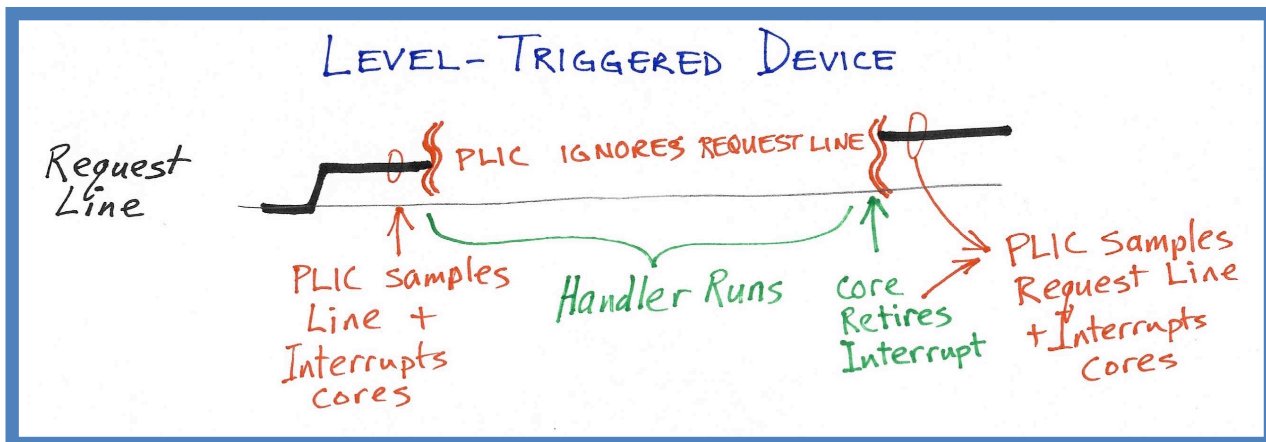
Level-Triggered
Edge-Triggered

The **EDGE_TRIGGERED_ARRAY** — which should be initialized at start-up — determines how the PLIC will sample that line. The array is stored in a single doubleword I/O register with one bit for each device. Bit k in the doubleword corresponds to device k. If the bit is initialized to 0, the line will be level-triggered; if the bit is 1, the line will be edge-triggered.

[7] Our discussion of a "single wire" is conceptual only. The actual circuitry by which the interrupt signal is carried from device to PLIC or PLIC to core may be more complex.

Level-Triggered

If the device operates as a level-triggered device, then the interrupt request line from the device to the PLIC is sampled periodically. If the line is high, the PLIC will begin operation and will send a "PLIC Interrupt" to all cores that have enabled that device. Then, some core will claim the interrupt, execute a handler, and ultimately retire the interrupt.



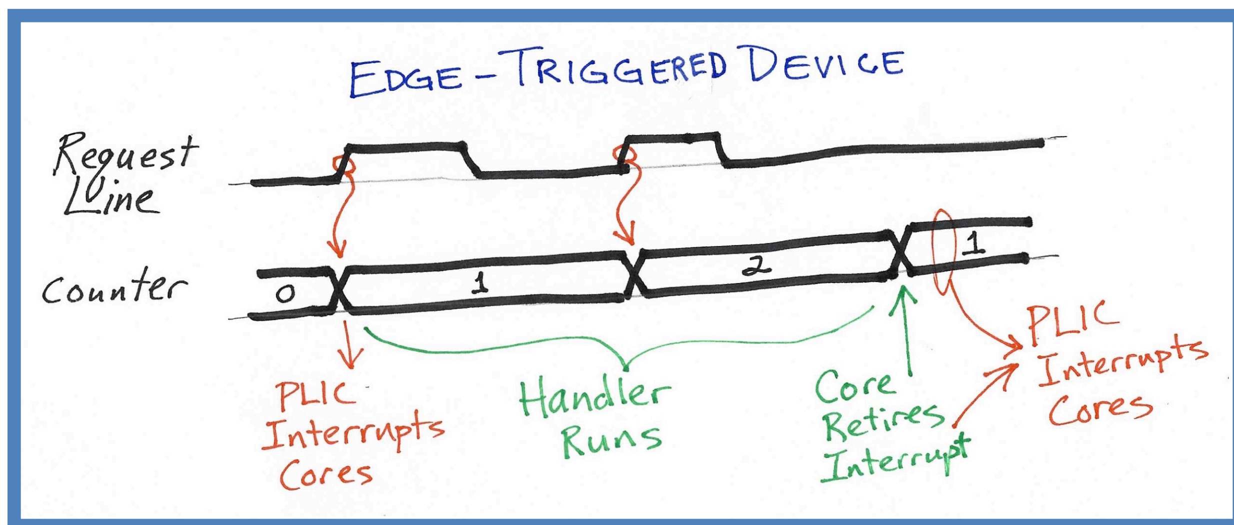
For a level-triggered device, the act of retiring the interrupt will cause the PLIC to once again sample the device's request line. If high, the process will repeat and another interrupt will be generated. If low, no interrupt will be generated. If there is no interrupt, the PLIC will continue to sample to line periodically, e.g., on every clock pulse. Whenever the line goes high again, the PLIC will start a new interrupt cycle.

For a level-triggered device, once an interrupt has occurred and the PLIC has notified all enabled cores, the PLIC will then ignore the interrupt line from that device. For level-triggered devices, it is expected that the device will keep its request line high for many clock cycles until some operation by the handler code resets the device, causing it to lower its request line.

Edge-Triggered

For edge-triggered devices, the device is expected to send a single pulse on its request line to the PLIC. It is the **rising edge on the request line** that will cause the PLIC to generate an interrupt in all enabled cores.

However, with an edge triggered device, the device can continue to generate more interrupts while the previous interrupt is being handled.



For each edge-triggered device, the PLIC maintains a counter. This counter is internal to the PLIC; it is not directly accessible by the cores and is initialized to zero on power-on-reset. Every time the PLIC detects a rising edge on the request line from a device, the counter for that device is incremented. Every time a core retires an interrupt, the counter is decremented.

There can only be at most one interrupt active for any given device. If an edge-triggered device has an interrupt active (i.e., that interrupt has not yet been retired) and the device raises subsequent interrupts, then the counter will be incremented, but no further interrupts to the cores will occur. When the current interrupt is retired, the PLIC will process subsequent interrupts and raise a new "PLIC Interrupt" in the enabled cores. When an interrupt for an edge-triggered device is retired, the PLIC will decrement the counter associated with that device. If, after the

decrement, the counter is still positive, then a new interrupt will be forwarded to the cores.[8]

Claiming Multiple Interrupts Not Allowed

Once a core has claimed an interrupt, the handler will normally run with interrupts disabled until the interrupt is retired. Interrupts should not be re-enabled until after the interrupt has been retired.

Once some core has claimed an interrupt from device K and before that interrupt has been retired, the PLIC will not raise another "PLIC Interrupt" at any core concerning device K, even if device K has requested another interrupt.
To restate this, consider a situation in which a core is handling an interrupt from device K and has not yet retired it. Now assume device K raises another interrupt before the core has retired the previous interrupt. The interrupt from device K will not be forwarded to that core or to any other core until the previous interrupt is retired.

If the device is "edge-triggered", the second interrupt will not be lost. Instead, it will remain pending because the counter will be incremented. The PLIC will signal a second interrupt only after the first interrupt is retired.

If the device is "level-triggered", this situation cannot arise; the level of the interrupt line from device K is only checked when the previous interrupt is retired. In other words, if the device is level-triggered, the interrupt request line from the device to the PLIC will be ignored until the previous interrupt is retired. Thus, the device cannot cause a second interrupt while the first interrupt is being handled. At the moment the first interrupt is retired, the request line from the device to the PLIC

[8] Edge-triggered devices that are integrated on the same chip as the PLIC and running within the same clock domain will presumably be synchronized with the PLIC circuitry. Like all on-chip signals, the interrupt request line from the device to the PLIC will be sampled on the rising edge of the clock. At each clock cycle, the counter will be incremented if and only if the request line is high. Thus, "edge-triggered" here means not so much as "the rising edge" but "high on a given clock cycle"; if the line happens to remain high for multiple clock cycles, the counter would be incremented on each clock cycle.

will be sampled. If the line is still high, the PLIC will dispatch the second interrupt to all cores for which that device is enabled.[9]

This means that interrupts from device K will be processed one-at-a-time, although each interrupt may be processed on a different core. And this precludes a core from handling more than one interrupt simultaneously.

> ***The PLIC serializes all interrupt handlers for a given device, and it serializes all interrupt handlers on a given core.***

After some core X has claimed an interrupt from device K but before it has retired that interrupt, other devices may request interrupts. If those other devices are also enabled for core X, then a "PLIC Interrupt" will be raised at core X. In other words, when a second device requests an interrupt, that interrupt will trigger interrupts at all cores, including X, which are enabled for that device. Presumably, core X will leave interrupts disabled until after it retires the interrupt from device K, so any subsequent interrupts to core X will remain pending.[10]

Discussion of Handler Approaches

In some operating systems, interrupts will be handled as follows:

- Device D requests an **interrupt**
- Core C is interrupted and interrupts are temporarily **disabled**
- The interrupt handler **claims** the interrupt for device D
- The interrupt handler **wakes up** a sleeping "device service process"
- The interrupt handler notifies the PLIC that the interrupt is **retired**.
- The handler **re-enables interrupts** and returns to other tasks
- Later, the "**device service process**" is scheduled and interacts with device D

But note that each interrupt must be retired before attempting to claim the next interrupt.

[9] The request line from a level-triggered device is also monitored when there are no interrupts active for that device. Typically it would be sampled on every edge of the PLIC's clock.

[10] At the PLIC implementor's option, if some device D raises an interrupt which is enabled at several cores and that interrupt is then claimed by some core, the PLIC may cancel the interrupt at the other cores, assuming there is no other device also interrupting them.

In the above scenario, the interrupt handler retires the interrupt before returning to other tasks. This means that the “device service process” runs after the interrupt has been retired, and at a time when future interrupts may be recognized and claimed. Thus, the device in question may generate additional interrupts (if its design allows for this) before the “device service process” has completed.

If devices can generate multiple interrupts between servicing, this could be a problem.

One solution is for the interrupt handler to fully service the interrupt and eliminate the need for any special "device service process” that is scheduled separately. For example, a UART might generate interrupts whenever a byte arrives on the receive channel. This could happen at any time. In this design, the interrupt handler will get each character, add it to a buffer, and retire the interrupt before re-enabling interrupts. This way, multiple input characters will each generate an interrupt and each character will be retrieved. Any tasks that are scheduled and run with interrupts enabled will only deal with the buffer and not care about interrupts or the device.

Another solution is for the interrupt handler to avoid retiring the interrupt, and leave that to the “device service process”. So the interrupt will be claimed by the interrupt handler, but it will not be retired until later, by the “device service process” whenever it is scheduled. Of course, this means the core will not be able to service any PLIC interrupt until after the “device service process” completes execution.

A third solution is make sure that each device will not generate the next interrupt until after the previous interrupt has been fully serviced. This might be typical of disks, Of course, a “read-sector” or “write-sector" command will interrupt when complete, but the disk will not then interrupt again until after the next command is issued. So it is safe for the interrupt handler to retire the interrupt before the “device service process” runs, since there cannot be a subsequent interrupt until after the “device service process” runs and issues the next command to the device.

Implementation in the Emulator

Next, we describe the implementation of the PLIC used in the Blitz-64 emulator.

This section can be safely skipped.

The following variables are used within the emulator:

EDGE_TRIGGERED_ARRAY — 64 ints, one per device
    0 = This device is level triggered
    1 = This device is edge triggered
ENABLE_ARRAY — 128 × 64 bits, one doubleword per core
    1 = This device can interrupt this core
    0 = This core will not get interrupts from this device
ENABLE_ARRAY_BY_DEVICE — 64 × 128 bits, two doublewords per device
    1 = This device can interrupt this core
    0 = This core will not get interrupts from this device
COUNTER — 64 ints, 1 per device
    0 = No interrupt (i.e., all previous interrupts have been claimed)
    n = There are *n* unclaimed interrupts for this device
PROCESSING — 128 ints, 1 per core
    -1 = This core is available to be interrupted
    k = This core has claimed (but not retired) an interrupt from device *k*
DEVICE_STATUS — 64 ints, 1 per device
    -1 = This device is not requesting an interrupt, or
        no core has claimed the current interrupt
    x = Core *x* has claimed (but not retired) an interrupt for this device
INITIALIZATION_COMPLETE — bool
    0 = No LOAD/STORE to CLAIM_ARRAY has yet occurred.
    1 = We have seen LOADs/STOREs to the CLAIM_ARRAY.

PLIC-related functions:

signalInterruptFromDevice (dev: int)
unsignalInterruptFromDevice (dev: int)
checkInterruptsForDevice (dev: int)
cancelUnneededInterrupts ()

On reset:

EDGE_TRIGGERED_ARRAY — Set to zeros
ENABLE_ARRAY — Set to zeros
ENABLE_ARRAY_BY_DEVICE — Set to zeros
COUNTER — Set to zeros
PROCESSING — All entries set to -1
DEVICE_STATUS — All entries set to -1
INITIALIZATION_COMPLETE — Set to 0

Whenever a LOAD or STORE to the **CLAIM** memory-mapped array is executed, **INITIALIZATION_COMPLETE** will be set to true. This flag is used to make sure the code doesn't try to modify **ENABLE_ARRAY** or the **EDGE_TRIGGERED_ARRAY** after kernel initialization is complete. If the kernel code misbehaves, a user warning is displayed.

A STORE to the **EDGE_TRIGGERED_ARRAY** memory-mapped doubleword is transformed into an initialization of the internal **EDGE_TRIGGERED_ARRAY**, which is more convenient to use. If **INITIALIZATION_COMPLETE** is true, then a user warning is displayed. Otherwise, no further action.

The **ENABLE_ARRAY** maps exactly to the memory-mapped registers of the same name. A STORE to any of these registers will update the corresponding array element directly. The **ENABLE_ARRAY_BY_DEVICE** contains the exact same information, except it is indexed by device number, rather than core number. Each and every STORE to the **ENABLE_ARRAY** will also cause the complete reinitialization of the **ENABLE_ARRAY_BY_DEVICE**. A STORE to **ENABLE_ARRAY** after **INITIALIZATION_COMPLETE** has been set will be flagged with a user warning.

The **ENABLE_ARRAY_BY_DEVICE** is not modified, except when the kernel writes to **ENABLE_ARRAY** during initialization. It is used by the PLIC-related code in the emulator, and the **ENABLE_ARRAY** is ignored after **ENABLE_ARRAY_BY_DEVICE** is initialized.

For edge-triggered devices, the following function should be invoked by the code associated with the device:

signalInterruptFromDevice (dev: int)

For level-triggered devices, this function should be invoked by the code associated with the device whenever the interrupt level is changed to HIGH. Whenever the device wishes to lower the interrupt request line, it calls:

unsignalInterruptFromDevice (dev: int)

The **signalInterruptFromDevice** function will check to see if the device is edge- or level-triggered. If it is edge-triggered, it will increment the corresponding **COUNTER** array element. If the device is level-triggered, it will set the element to 1. In either case, it will then call:

checkInterruptsForDevice (dev: int)

Whenever **unsignalInterruptFromDevice** is invoked, the corresponding **COUNTER** array element will be set to 0. We may also check that the device is truly level-triggered and issue a user warning if not. It will then call the following function to adjust things as needed:

Reset_PLIC_Interrupt_Pending ()

This function will cancel all pending PLIC Interrupts. It will then go through **COUNTER** array to identify every device requesting an interrupt and, if the device is not currently claimed by any core (i.e., **DEVICE_STATUS** == -1), it will raise a PLIC Interrupt for all enabled cores. There is a **PLIC_Interrupt_Pending** flag for each core, and an interrupt is signaled by setting this flag[11]. It will consult **ENABLE_ARRAY_BY_DEVICE** to determine which cores are to be interrupted.

The **checkInterruptsForDevice** function will consult the **COUNTER** array to determine whether this device is needing an interrupt and the **DEVICE_STATUS** array to determine whether there is an ongoing (i.e., claimed but not retired) handler working on this device. If there is an interrupt required, but none in progress, a PLIC Interrupt will be raised at all enabled cores, which can be determined by consulting the **ENABLE_ARRAY_BY_DEVICE** array.

[11] The **PLIC_Interrupt_Pending** flag is checked before each instruction and a trap occurs if it is set and interrupts are enabled.

Whenever a LOAD from the **CLAIM_ARRAY** occurs, an interrupt is being **claimed**. First, we check to make sure this core doesn't have any interrupts in progress by checking the **PROCESSING** array. Then we search for a device such that (1) it has outstanding unclaimed interrupts (the **COUNTER** array), (2) no core is currently servicing an interrupt from this device (the **DEVICE_STATUS** array), and (3) this device is enabled for the requesting core (the **ENABLE_ARRAY_BY_DEVICE** array). If no such device is found, -1 is returned as the result of the LOAD. Otherwise, we decrement the **COUNTER** array, set the **DEVICE_STATUS** array for this device to the current core, and set the **PROCESSING** array for this core to this device. Now, since this core has claimed the interrupt for this device, we adjust the **PLIC_Interrupt_Pending** flags for the other cores by calling **Reset_PLIC_Interrupt_Pending**.

Whenever a STORE into the **CLAIM_ARRAY** occurs, an interrupt is being **retired**. First, we make sure there is an outstanding (i.e., un-retired) interrupt for this core by checking the **PROCESSING** array, which indicates which device was being handled. Then we change the entry for this core to -1 and we also change the **DEVICE_STATUS** array for that device to -1. Then we ask whether there are still more interrupts from this device by checking the **COUNTER** array for this device. If so, we raise the **PLIC_Interrupt_Pending** flag for all cores that are enabled for this device. Finally, we ask whether there are still unclaimed interrupts for devices for which this core is enabled, If so, we raise the **PLIC_Interrupt_Pending** flag for this core.

Interrupt Priorities

It is possible that the following interrupts will occur simultaneously:

Timer Interrupt ← *highest*
DMA Complete Interrupt
PLIC Interrupt ← *lowest*

Only one interrupt will be processed at a time. In other words, when interrupts are enabled, one interrupt will be selected and a trap will occur for that interrupt. Any other pending interrupts will remain pending until interrupts are again re-enabled.

The Timer Interrupt will be given precedence, with the others in turn. This is how the emulator orders priority; other implementations may order the priority of interrupts differently.

Chapter 3: UART0

Quick Summary

- A UART Communication Device
  - One **Tx** and one **Rx** Channel
- Interface is through memory-mapped I/O registers
  - **SEND_BYTE**
  - **RECV_BYTE**
  - **STATUS**
    - **RECV_READY** bit 0
    - **SEND_READY** bit 1
  - **SETUP**
    - **INTERRUPTS_REQUESTED** bit 0
- Optional interrupts to PLIC
  - When the device becomes free to send a byte
  - When the device has received another byte
- The Blitz-64 emulator implements this device

Introduction

This device is a **Universal Asynchronous Receive Transmit (UART)** channel, sometimes known as a "serial port". There is a transmit channel for transmit and another channel for receive. Bytes can be sent along each channel independently at the same time, i.e., the channel is full duplex. A complete byte at a time is sent.

In hardware, the byte is linearized and sent to some other device one bit at a time. At the minimum the hardware uses three wires:

Tx
Rx
Gnd

When emulated, the transmission may happen in other ways.

In this chapter, we describe what is implemented by **the Blitz-64 emulator**; other implementations may vary. Variations may include:

- Multiple channels, instead of a single Tx/Rx channel
- Ability to control the baud rate, the parity bit, and the number of stop bits
- Different register mappings and meanings[12]

Memory-Mapped I/O Registers

The following four **hardware registers** are implemented and these are memory-mapped into locations in the physical address space.

SEND_BYTE — Doubleword, write only
Only the lower 8 bits are used, higher bits are ignored.

RECV_BYTE — Doubleword, read only
The value will be 0 … 255, i.e.,
`0x0000_0000_0000_0000 … 0x0000_0000_0000_00ff`

STATUS — Doubleword, read only
RECV_READY (bit 0): 0=no recv byte is available; 1=recv byte is available
SEND_READY (bit 1): 0=send channel is busy; 1=send channel is ready
Other bits are zero, but should be ignored.

SETUP — Doubleword, write only
INTERRUPTS_REQUESTED (bit 0): 0=do not interrupt; 1=cause interrupts
Other bits are ignored; reserved for future baud/stop/parity configuration

[12] There are a number of existing chips and protocols for UART devices which are in widespread use. While we choose to use a simple, minimal design, other Blitz-64 implementations may choose to mimic those legacy systems.

These doublewords are located at the following offsets within the page allocated to the UART0 device:

Offset	Size in bytes	Register	
0	8	**SEND_BYTE**	*write-only*
0	8	**RECV_BYTE**	*read-only*
8	8	**STATUS**	*read-only*
8	8	**SETUP**	*write-only*

Note that the same address can be used for multiple registers. For example, a LOAD from offset 8 returns the status word, while a STORE to that same address will perform a setup operation.

Operation

Before usage, the SETUP register should be written to. Bit 0 — the **INTERRUPTS_REQUESTED** bit — determines whether the device will cause an interrupt or not.

If a 0 is written, no interrupts will be generated. To use the device, the code should check the **STATUS** register to determine when bytes can be safely read from **RECV_BYTE** or written to **SEND_BYTE**.

If **INTERRUPTS_REQUESTED** is 1, the device will interrupt the core(s) whenever either

- A byte has been received on the receive channel
- The send channel can accept the next byte to transmit

The interrupt will be sent to the PLIC, which will then forward it to whichever cores have enabled interrupts from the UART0 device.

Additional bits of the **SETUP** register may be defined later. These bits might be used to configure the UART circuitry's

- Baud Rate
- Number of bits (7 or 8) by "byte" transmitted
- Number of stop bits
- Parity bit (none, even, or odd)

The emulator simulates hardware so these parameters are neither relevant nor needed.

To send a byte, the core will store a value into the **SEND_BYTE** register. Only the lower 8 bits are sent; the upper 56 bits are ignored. If the UART0 device is not ready to transmit — that is, if the **SEND_READY** status bit is not 1 and the device is still busy sending the previous byte — the results are undefined and the byte will be lost or perhaps some garbage will be transmitted on the **Tx** line. If the **INTERRUPTS_REQUESTED** control bit was set to 1, the device will generate an interrupt after the byte has finished sending and the device is ready for the next byte to be written into **SEND_BYTE**.

Whenever a value is STOREd into **SEND_BYTE**, the **SEND_READY** status bit will immediately go to 0 and will remain 0 until the device has finished sending a byte and is ready to send the next byte.

To receive a byte, the core must wait for incoming data. When the UART0 device receives a byte, it will drive the **RECV_READY** status bit to 1. It will also generate an interrupt if the **INTERRUPTS_REQUESTED** control bit was set to 1. After that, the core can safely LOAD from the **RECV_BYTE** register. The byte that was received will be returned as an integer in the range 0…255. The register is a doubleword and the upper 56 bits will be zeros. A LOAD operation before **RECV_READY** goes to 1 is undefined and may return anything, such as a copy of the previously received byte.

Whenever a byte is LOADed from **RECV_BYTE**, the **RECV_READY** status bit will immediately go to 0 and will remain 0 until a new byte is received over the **Rx** line.

Notes

This device is named “UART0” — instead of simply “UART” — to emphasize that it is about the simplest interface to a UART device possible. It is implemented by the Blitz-64 emulator as described. The UART0 interface is unique to Blitz-64, but it is quite similar to other UART interfaces.

Hardware implementations may use the UART0 interface as described here, or may use something a little different. For example, future hardware may define the memory-mapped I/O registers to map more closely to the so-called “16550 UART”

interface. Different specifications for a UART device interface should be named something besides “UART0”.

The PLIC handles and dispatches interrupts from devices numbered 0…63. In the Blitz-64 emulator, the PLIC maps the UART0 device to device number 0. The use of 0 in both is coincidence.

Chapter 4: DISK0

Quick Summary

• A Disk Device
  — Providing Sector Read and Write Operations
• Interface is through memory-mapped I/O registers
  — **STATUS**
    — **BUSY** bit 0: 1=operation in progress; 0=idle and free
    — **ERROR** bit 1: 1=operation failed; 0=operation succeeded
  — **SETUP**
    — **INTERRUPTS_REQUESTED** bit 0: 1=interrupt; 0=do not interrupt
  — **SECTOR_START**
  — **SECTOR_COUNT**
  — **MEMORY_ADDRESS**
  — **COMMAND**
• Optional interrupts to PLIC
  — Interrupt upon command completion (failure or success)
• The emulator implements this interface
  — Disk is emulated using a file on the host system
  — Disk filename provided on emulator command line
  — Emulation Parameters
      Sector size (default is 512 bytes)
      Disk size (default is 2000 sectors)

Introduction

This device simulates some form of **long-term stable storage**, such as a disk or flash memory. The design is intended to mimic real hardware, although the emulator will use a file on the host system to store the data.

This device responds to **two commands: READ and WRITE**. Each command is passed:

- A “disk address”, i.e., where the data is stored or retrieved
- The length in bytes of the data to be transferred
- An address in memory, i.e., where the data is read or written

The READ command will transfer data from the stable storage into memory. The WRITE command will transfer data from the memory to the stable storage.

The device described here is only present in an emulated system. The stable storage will be **backed by a file on the host computer system**.

The device implemented by the emulator is sufficient to support an OS running on Blitz, and has been used to store a complete Unix file system for a Blitz implementation of the xv6 OS.

It should be noted that real devices are both different in detail and more complex in operation. For example, a real device may encounter errors so some operations may fail or time-out. The emulator models the delay associated with operations. READs and WRITEs are not instantaneous, but real systems (such as rotating disks) will exhibit delays that depend on details such as track, head location, and rotation which are not modeled by the emulator.

Memory-Mapped I/O Registers

STATUS — Doubleword, read only
Only the lower 2 bits are used, higher bits are zeros.
— **BUSY** (bit 0): 1=operation in progress; 0=idle and free
— **ERROR** (bit 1): 1=operation failed; 0=operation succeeded

SETUP — Doubleword, write only
Only the lower bit is used, higher bits are ignored.
— **INTERRUPTS_REQUESTED** (bit 0):
1=interrupt when operation completes; 0=do not interrupt

SECTOR_START — Doubleword, write only

SECTOR_COUNT — Doubleword, write only

MEMORY_ADDRESS — Doubleword, write only

COMMAND — Doubleword, write only
0 = perform a read, transferring sectors from disk to memory
1 = perform a write, transferring sectors from memory to disk

The registers are located within a memory-mapped I/O page at these offsets:

Offset	Size in bytes	Register	
0x0000	8	**STATUS**	*read-only*
0x0000	8	**SETUP**	*write-only*
0x0008	8	**SECTOR_START**	*write-only*
0x0010	8	**SECTOR_COUNT**	*write-only*
0x0018	8	**MEMORY_ADDRESS**	*write-only*
0x0020	8	**COMMAND**	*write-only*

Operation

Before using the disk, the program should write into the **SETUP** register either a 0 or a 1. If 1 is written, then the device will interrupt the core(s) whenever an operation completes. The interrupt will be channeled through the **Platform-Level Interrupt Controller (PLIC),** which will route the interrupt to the cores, as described elsewhere in this document.

At any one moment in time, the disk is either busy with a command or idle. To determine the status of the disk, the code should LOAD from the **STATUS** word. The **ERROR** bit will reflect the result of the previous operation. (It is undefined before the first command is issued.) The **BUSY** bit will be set to 1 if the desk is still working on the previous command.

The disk can perform only two operations: read and write. To perform a read or write operation, the core must:

(1) LOAD the **SECTOR_START** register with an integer to indicate the number of the first sector to be transferred. This number should range between 0 and **NUMBER_OF_SECTORS**-1.

(2) LOAD the **SECTOR_COUNT** register with the number of sectors to be transferred. This should be a small integer, greater than 0.

(3) LOAD the **MEMORY_ADDRESS** register with a valid physical address. This address should be doubleword aligned.

(4) LOAD the **COMMAND** register with either 0 or 1, to indicate a read (0) or a write (1) is to be performed.

Operations (1), (2), and (3) above may be done in any order. They may also be skipped; if skipped, the register will retain its previous value. There will be no error reporting or interrupts as a result of LOADing these registers. The registers should not be loaded while the disk is BUSY with an operation.

LOAD (4) must be done after the other three registers have been loaded. Step (4) will begin the operation.

If the disk is busy at the moment COMMAND is LOADed, the command will be ignored and the previous operation will continue uninterrupted. The emulator will detect this and halt execution.

Each operation (read or write) will copy one or more sectors between memory and the disk. “Write” moves data from memory to disk and “read” moves data from the disk to memory.

The number sectors to be moved must all be valid, i.e., within the size of the disk. For example, if the disk contains 2,000 sectors, a read of 2 sectors from 1,998 is okay and will read the last two sectors (1,998 and 1,999), but a read of 3 sectors from this same **SECTOR_START** will result in an error.

When an error is detected, it is undefined whether some sectors may have been transferred. (The emulator will detect this error before transferring any data and immediately cause a halt to emulation.)

When an error is detected, the **ERROR** bit in the **STATUS** word will be set, and the **BUSY** bit will be cleared. If **INTERRUPTS_REQUESTED** is true, an interrupt will also

be signaled. The **STATUS** will then remain unchanged until **COMMAND** is again LOADed.

The **SECTOR_COUNT** determines the number of bytes to be transferred:

number_of_bytes_to_copy = **SECTOR_COUNT** × **SECTOR_SIZE**

where **SECTOR_SIZE** is an emulation parameter.

The **MEMORY_ADDRESS** register, along with the number_of_bytes_to_copy determines where in memory the data is to be copied to/from. This should be a valid address in physical memory. **MEMORY_ADDRESS** should be a 44 bit physical address. There is no participation with virtual memory.

If the memory address range is illegal, the operation is considered to be incorrect and may—or may not—cause an error. In the current emulator, an attempt to access bytes beyond the physical memory will be detected by the emulator, and emulation will immediately halt.

MEMORY_ADDRESS, **SECTOR_COUNT**, and **SECTOR_START** are interpreted as positive integers. A **SECTOR_COUNT** of 0 is considered incorrect. The emulator will catch this an immediately halt emulation.

The following are the possible error conditions:

Emulator suspends emulation:
- **SECTOR_COUNT** = 0
- **SECTOR_START** + **SECTOR_COUNT** >= **NUMBER_OF_SECTORS**
- **MEMORY_ADDRESS** + number_of_bytes_to_copy ≥ physical_memory_size
- **MEMORY_ADDRESS** is not doubleword aligned.
- LOADing **SECTOR_START**, **SECTOR_COUNT**, **MEMORY_ADDRESS**, **or COMMAND** while disk **STATUS** is **BUSY**
- Host error, problems with the host file

Emulator does not implement:
- Simulated disk errors (transient or non-recoverable)

Emulator Implementation

The emulator uses the following "emulation parameters". (These are defaults which can be changed by editing the file named **emulationParms**.)

	default value
DISK0_SECTOR_SIZE	512
DISK0_NUMBER_OF_SECTORS	2000
DISK0_DEVICE_START_ADDR	0x4_0020_8000
DISK0_OPERATION_DELAY	10000

The emulator (a program named "**blitz**") is run from the command line in a Unix/Linux/Mac shell. One option is the name of a file to use as the "disk image file". The option is specified as in:

```
% blitz … -disk MyImageFile.img
```

The file is not opened until a DISK0 command is issued.

If the file does not exist, it is created and set to the given size. The emulator may print a message and suspend emulation, but you can type "go" to continue execution if you are okay with the file being created.

If the file exists, but is larger than indicated by the emulation parameters, the tail of the file is ignored.

If the file exists, but is shorter than indicated by the emulation parameters, the file is immediately enlarged to the expected size.

The file is closed when the emulator exits or when the "reset" or "rerun" commands are used.

When a disk operation is performed, there will be a delay after the **COMMAND** register is written to and the moment that the **BUSY** bit in the **STATUS** word will changed to 1 and an interrupt will be signaled (if requested). This delay is determined by the emulation parameter **DISK0_OPERATION_DELAY**, which is given as a number of instructions.

If you are emulating a multi-core processor, keep in mind that the emulator will execute some instructions for core 0, followed by some instructions for core 1, etc.

For example, a delay of 10,000 instructions on a 10-core system means that each core will execute 1,000 instructions on average before the disk operation is completed.

Notes

The PLIC handles and dispatches interrupts from devices numbered 0…63. In the Blitz-64 emulator, the PLIC maps the DISK0 device to device number 1.

Chapter 5: DMA Controller

Quick Summary

- Direct Memory Access
  - — Move a large block of memory
  - — Zero a large block of memory
- Includes **crypto-engine** functions
  - — Perform secure hashing (using SHA-256)
  - — Perform AES encryption and decryption

Background

In general terms, a **Direct Memory Access** (DMA) controller is capable of moving large blocks of data from one location in the physical address space to another location. This includes both installed physical memory and the memory-mapped I/O device region.

Such operations are useful in moving sectors/pages/blocks both to and from I/O device buffers. A DMA controller can also be used to copy pages from one address space to another (e.g., to duplicate a copy-on-write page). The DMA controller can also be used to zero-out memory pages, which may be necessary for newly allocated pages to prevent information leakage from one address space into an unrelated address space.

Of course these data moving tasks can be done directly by the core. However, this may not be the best approach, since the core will not be usable during the operation. Furthermore, since an instruction loop is required, copying by the core will be relatively slow. The DMA controller avoids instruction execution and performs the repetitive LOAD-STORE cycle directly in hardware, which can drive the memory bus at its maximum bandwidth.

Generally speaking, a DMA controller will interleave accesses to the memory bus with the accesses being made by the core, to avoid locking up the core. The presence of DMA activity may slow the core, since LOADs, STOREs, and FETCHes that cannot be served by caches may have increased latency times due to bus contention. However, the core will continue to operate during DMA operations, freeing the core to do things the DMA controller cannot do.

A DMA controller is said to be "programmed" to performed a task. The DMA controller is commanded by the core to perform a task and, when complete, it signals an interrupt to the core. By "programmed" we mean that the DMA controller is issued a command or series of commands. These commands are given by writing predetermined values to predetermined words within the memory-mapped region occupied by the DMA controller.

The Blitz DMA Controller

In this chapter we describe a **Direct Memory Access** (DMA) controller is capable of the following tasks:

- Move a large block of memory
- Zero a large block of memory
- Perform secure hashing (using SHA-256)
- Perform AES encryption and decryption

As of this writing, the device described here is implemented in the emulator. It is not yet implemented in VLSI.

This device can perform one task at a time and is either "busy" or "free". There is no queue of waiting tasks.

A "**device register**" is doubleword in the DMA controller's page. Each "register" is 64 bits and is located at a doubleword aligned address. The device is controlled by storing into "registers" and the results are obtained by reading from the "registers".

Here are the device registers:

Offset (hex)	Offset (dec)		Register Name
0000	0	write-only	DMA_COMMAND
0008	8	r/o	DMA_STATUS
0010	16	write-only	DMA_START_ADDR
0018	24	write-only	DMA_TARGET_ADDR
0020	32	write-only	DMA_BYTECOUNT
0028	40	r/o	DMA_SHA256_0
0030	48	r/o	DMA_SHA256_1
0038	56	r/o	DMA_SHA256_2
0040	64	r/o	DMA_SHA256_3
0048	72	write-only	DMA_AES_KEY_0
0050	80	write-only	DMA_AES_KEY_1
0058	88	write-only	DMA_AES_KEY_2
0060	96	write-only	DMA_AES_KEY_3

Additional functionality may be added in the future; additional registers will be defined to control such enhancements at that time.

The arguments (such as “starting address”, “byte count”, and so on) should be stored first, in any order. The task is initiated by writing a command code into the DMA_Command register.[13]

Upon completion of the task, the DMA controller will interrupt the core. In addition, a status code will be available in the “DMA_STATUS” register.

[13] The registers should not be written while the device is busy; if so, the behavior is undefined and considered to be an error. The status register “DMA_STATUS” may be read at any time. Any attempt to read the other registers when the device is busy is undefined and considered to be an error. Any attempt to write to a read-only register, or read from a write-only register, is undefined and considered to be an error. Any attempt to read or write to an undefined address within the page is undefined and considered to be an error. All registers are doublewords; any attempt to read individual bytes, halfwords, or words is undefined and considered to be an error.

For reference, here are the command codes:

hex	decimal	command	
0001	1	DMA_MOVE	Move memory
0002	2	DMA_ZERO	Zero memory
0003	3	DMA_SHA256_SIMPLE	SHA256 (Simple, only one chunk)
0004	4	DMA_SHA256_INITIALIZE	SHA256 (Initialize)
0005	5	DMA_SHA256_CHUNK	SHA256 (Process next chunk)
0006	6	DMA_SHA256_FINALIZE	SHA256 (Finalize)
0007	7	DMA_AES256_PREPARE	AES-256 Prepare Key
0008	8	DMA_AES256_EN_SIMPLE	AES-256 Encrypt (Simple)
0009	9	DMA_AES256_EN_INITIAL	AES-256 Encrypt (Initial segment)
000a	10	DMA_AES256_EN_MIDDLE	AES-256 Encrypt (Middle segments)
000b	11	DMA_AES256_EN_FINAL	AES-256 Encrypt (Final segment)
000c	12	DMA_AES256_DE_SIMPLE	AES-256 Decrypt (Simple)
000d	13	DMA_AES256_DE_INITIAL	AES-256 Decrypt (Initial segment)
000e	14	DMA_AES256_DE_MIDDLE	AES-256 Decrypt (Middle segments)
000f	15	DMA_AES256_DE_FINAL	AES-256 Decrypt (Final segment)

For reference, here are the status codes:

hex	decimal	command	
0000	0	DMA_OK	Last operator completed
0001	1	DMA_BUSY	Operation in progress

The addresses (i.e., DMA_START_ADDR and DMA_TARGET_ADDR) are physical addresses and should lie within 0x0_0000_0008 and 0x7_FFFF_FFFF. They must not be virtual addresses and paging will not be involved.[14]

Normally the addresses will lie in physical RAM, but they may also include ROM, Secure Storage, and FLASH memory.[15]

[14] Addresses are 35 bits, with the "physical/virtual" bit assumed to be 0.

[15] Exactly which Memory-Mapped I/O devices can be operated on by the DMA controller depends on what devices are present and is therefore implementation-dependent. But the DMA controller must be able to operate on anything that is "memory-like", since the compiler may generate code using the DMA controller to access such memory. In particular, the security-related functionality will certainly be applied to the ROM and SecureStorage devices.

Moving Blocks of Memory

To move a block of memory:

STORE an address into **DMA_START_ADDR**
STORE an address into **DMA_TARGET_ADDR**
STORE an integer into **DMA_BYTECOUNT**
STORE the command **DMA_MOVE** into **DMA_COMMAND**
Wait for the task to complete

This operation is primarily intended for copying entire 16 KiByte pages, to support things like copy-on-write sharing and moving address space pages from private to shared memory.

The addresses should be doubleword aligned and the number of bytes to be moved should be a multiple of 8. The last 3 bits of DMA_START_ADDR, DMA_TARGET_ADDR, and DMA_BYTECOUNT are ignored.

The blocks of memory should not overlap; if so the result is undefined.

Wait for Task to Complete

Regardless of the command, when the DMA controller completes a task, it will cause an interrupt. The status doubleword DMA_STATUS can be read at any time and will tell whether the DMA controller is busy or ready to receive another command.

To wait for a task, the program might chose to do a busy-loop, repeatedly querying DMA_STATUS. However, this may increase bus traffic and/or slow the DMA controller down, as well as waste cycles, so this approach is not recommended unless you know for sure the wait will be short.

The other approach is to proceed to other tasks and wait for the interrupt to trigger further action. It is envisioned that a two Semaphores will protect the DMA controller. Semaphore #1 will be used to make sure only one thread is using the DMA controller at a time. Semaphore #2 will be used to signal the interrupt.

Semaphore #1 will act as a “mutex” lock, allowing only one thread at a time to use the DMA device. Before using the DMA device, every thread must “wait” on

Semaphore #1 (i.e., the "down" or "P" operation). After the task is complete and the results have been retrieved from the device, the thread must "signal" the semaphore (i.e., the "up" or "V" operation), making the DMA device free and available to other threads.

Semaphore #2 is used to communicate the interrupt. When a thread which is using the DMA device is ready to wait for the completion of the task, it will "wait" on Semaphore #2. The interrupt handler will respond to the interrupt by "signaling" Semaphore#2, thus waking up the thread up. The thread should then retrieve the results and signal Semaphore #1.[16]

Zeroing Blocks of Memory

To zero a block of memory:

> STORE an address into **DMA_START_ADDR**
> STORE an integer into **DMA_BYTECOUNT**
> STORE the command **DMA_ZERO** into **DMA_COMMAND**
> Wait for the task to complete

This operation is primarily intended for clearing entire 16 KiByte pages, when processes terminate and address space pages are recycled.

The addresses should be doubleword aligned and the number of bytes to be zeroed should be a multiple of 8. The last 3 bits of DMA_START_ADDR and DMA_BYTECOUNT are ignored.

Note: This command uses DMA_START_ADDR and not DMA_TARGET_ADDR.

[16] With this approach, any thread which sends a command to the DMA device must always wait on Semaphore #2, or else signals from previous tasks will accumulate and prematurely terminate future unsuspecting threads. Furthermore, that wait must occur before Semaphore #1 is signaled. If there are some situations where some threads using the DMA controller will not be waiting, then an alternate, more complex design will be required.

SHA-256

A block of bytes (the "message") can be processed to yield a hash value, using the SHA-256 algorithm. The result of this operation is a 256 bit (i.e., 4 doublewords, or 4 × 64 bits) hash value.

We say that a region of memory is "**continuous**" if a single starting address and byte count suffice to locate the region in memory. If the region happens to span multiple pages, then all those pages must be adjacent and sequential. In other words, no gaps or jumping around is allowed.

Alternatively, a block of bytes could originate from a virtual address space. While it might be continuous in the virtual address space, it might happen to cross page boundaries. However, the DMA controller works only on physical addresses. While the block of bytes is continuous in the virtual address space, it may not be continuous in physical memory. Such a block must be broken into a sequence of two or more "**chunks**". Each chuck must be entirely continuous and can therefore be described with a starting address and byte count.

To compute the hash of a single, fully continuous block of memory:

> STORE an address into **DMA_START_ADDR**
> STORE an integer into **DMA_BYTECOUNT**
> STORE the command **DMA_SHA256_SIMPLE** into **DMA_COMMAND**
> Wait for the task to complete
> READ the 256 bit (i.e., 4 doubleword, or 4 × 64 bits) hash value from
>   **DMA_SHA256_0** … **DMA_SHA256_3**.[17]

The **DMA_START_ADDR** must be doubleword aligned, but the **DMA_BYTECOUNT** does not need to be a multiple of 8.

On the other hand, it may be that a User Mode process has requested the SHA-256 hash for a block of message bytes in a virtual address space and the block of message bytes crosses one or more page boundaries. In this case, the message must divided into chunks of bytes where each chunk lies wholly within a continuous range of physical memory.

[17] If you did not even wonder about most-significant/least-significant order, then you have escaped the mental contamination of Little Endian dementia.

The individual chunks may be any length; they do not need to be a multiple of 8 bytes.

The SHA-256 algorithm involves an initialization phase and a finalization phase. Here is the procedure:

For the first chunk:
  STORE the command **DMA_SHA256_INITIALIZE** into **DMA_COMMAND**
  Wait for the task to complete
For each chunk:
  STORE an address into **DMA_START_ADDR**
  STORE an integer into **DMA_BYTECOUNT**
  STORE the command **DMA_SHA256_CHUNK** into **DMA_COMMAND**
  Wait for the task to complete
After the last chunk:
  STORE the command **DMA_SHA256_FINALIZE** into **DMA_COMMAND**
  Wait for the task to complete
  READ the 256 bit (i.e., 4 doubleword, or 4 × 64 bits) hash value from
    DMA_SHA256_0 … **DMA_SHA256_3**.

The **DMA_START_ADDR** must be doubleword aligned, but the **DMA_BYTECOUNT** does not need to be a multiple of 8.

AES-256

The AES-256 algorithm uses a 256 bit (i.e., 4 doubleword, or 4 × 64 bits) key to either encrypt a message or decrypt a message. Since the algorithm is symmetric, the same key is used for both encryption and decryption. However, the encryption algorithm is different from the decryption algorithm.

The DMA controller will process a message and produce a result. The "source region" is the block of memory bytes containing the message to be processed. The "target region" is the block of memory bytes where the result of the encryption or decryption will be placed.

A region of memory may or may not be continuous.

We say that a region of memory is "**continuous**" if a single starting address and byte count suffice to locate the region in memory. If the region happens to span multiple pages, then all those pages must be adjacent and sequential. In other words, no gaps or jumping around is allowed.

Alternatively, a block of bytes could originate from a virtual address space. While it might be continuous in the virtual address space, it might happen to cross page boundaries. However, the DMA controller works only on physical addresses. While the block of bytes is continuous in the virtual address space, it may not be continuous in physical memory. Such a block must be broken into a sequence of two or more "**chunks**". Each chuck must be entirely continuous and can therefore be described with a starting address and byte count.

When both the source and target regions consist of a single chunk, we have a "simple" case.

To encrypt a "simple" continuous block of memory using AES-256:

STORE the key into **DMA_AES_KEY_0** … **DMA_AES_KEY_3**
STORE the command **DMA_AES256_PREPARE** into **DMA_Command**
Wait for the task to complete
STORE an address into **DMA_START_ADDR** (where to find the plaintext)
STORE an address into **DMA_TARGET_ADDR** (where to store the cipher-text)
STORE an integer into **DMA_BYTECOUNT**
STORE the command **DMA_AES256_EN_SIMPLE** into **DMA_Command**
Wait for the task to complete
Retrieve the cipher-text from the target area.

To decrypt a "simple" continuous block of memory using AES-256:

STORE the key into **DMA_AES_KEY_0** … **DMA_AES_KEY_3**
STORE the command **DMA_AES256_PREPARE** into **DMA_COMMAND**
Wait for the task to complete
STORE an address into **DMA_START_ADDR** (where to find the cipher-text)
STORE an address into **DMA_TARGET_ADDR** (where to store the plaintext)
STORE an integer into **DMA_BYTECOUNT**
STORE the command **DMA_AES256_DE_SIMPLE** into **DMA_COMMAND**
Wait for the task to complete
Retrieve the plaintext from the target area.

An AES-256 key is 256 bits (i.e., 32 bytes = 4 doublewords). Before any encryption or decryption, the key must be stored in the following DMA registers:

DMA_AES_KEY_0
DMA_AES_KEY_1
DMA_AES_KEY_2
DMA_AES_KEY_3

This key must be prepared before use.[18] The **DMA_AES256_PREPARE** command will convert the key into an internal representation, which will be stored in the DMA controller. This internal state will be used for future AES-256 encryptions and decryptions.

The same key may be used for multiple encryption and decryption operations and only needs to be prepared once. In other words, loading the **DMA_AES_KEY_0** … **DMA_AES_KEY_3** registers and executing the **DMA_AES256_PREPARE** command are performed first, and need not be repeated if the same key is used for multiple encryption/decryption operations.

For all AES-256 commands, the addresses **DMA_START_ADDR** and **DMA_TARGET_ADDR** must be doubleword aligned.

The AES algorithm encrypts and decrypts in units of 16 bytes (i.e., 128 bits) so the **DMA_BYTECOUNT** must be a multiple of 16. This means the message to be encrypted must be padded out to a multiple of 16 bytes and that any message to be decrypted will be a multiple of 16 bytes in length.

To encrypt a non-continuous block of memory using AES-256, the source and the target regions must be broken into a set of chunks, where each chunk is continuous and a multiple of 16 bytes in length. The first chunk must be be encrypted with the **DMA_AES256_EN_INITIAL** command. The last chunk must be encrypted with the **DMA_AES256_EN_FINAL** command. The middle chunks (if any), which lie between the initial and final chunks, are processed with a series of **DMA_AES256_EN_MIDDLE** commands.

[18] Before any encryption or decryption, the key must be "expanded" into something called the "round key", which is denoted "w". The step takes the 8 word (8 × 32 = 256 bit) key and initializes "w" which is another 56 words (i.e., Nb × Nr words, where Nb = number of words per block = 4, and Nr = number of rounds = 14)

Here is the sequence. As mentioned, the preparation of the key can be skipped if same key as used previously is to be used.

Prepare the key (optional, if same key as last time):
- STORE the key into **DMA_AES_KEY_0** … **DMA_AES_KEY_3**
- STORE the command **DMA_AES256_PREPARE** into **DMA_Command**
- Wait for the task to complete

For the first chunk:
- STORE an address into **DMA_START_ADDR** (where to find the plaintext)
- STORE an address into **DMA_TARGET_ADDR** (where to store the cipher-text)
- STORE an integer into **DMA_BYTECOUNT**
- STORE the command **DMA_AES256_EN_INITIAL** into **DMA_Command**
- Wait for the task to complete

For the each additional chunk:
- STORE an address into **DMA_START_ADDR** (where to find the plaintext)
- STORE an address into **DMA_TARGET_ADDR** (where to store the cipher-text)
- STORE an integer into **DMA_BYTECOUNT**
- STORE the command **DMA_AES256_EN_MIDDLE** into **DMA_Command**
- Wait for the task to complete

For the the final chunk:
- STORE an address into **DMA_START_ADDR** (where to find the plaintext)
- STORE an address into **DMA_TARGET_ADDR** (where to store the cipher-text)
- STORE an integer into **DMA_BYTECOUNT**
- STORE the command **DMA_AES256_EN_FINAL** into **DMA_Command**
- Wait for the task to complete

Retrieve the cipher-text from the target area.

The process for decryption is identical, except

instead of	*use*
DMA_AES256_EN_INITIAL	**DMA_AES256_DE_INITIAL**
DMA_AES256_EN_MIDDLE	**DMA_AES256_DE_MIDDLE**
DMA_AES256_EN_FINAL	**DMA_AES256_DE_FINAL**

To be precise, here is the procedure to decrypt a series of chunks:

Prepare the key (optional, if same key as last time):
- STORE the key into **DMA_AES_KEY_0** … **DMA_AES_KEY_3**
- STORE the command **DMA_AES256_PREPARE** into **DMA_Command**
- Wait for the task to complete

For the first chunk:
STORE an address into **DMA_START_ADDR** (where to find the cipher-text)
STORE an address into **DMA_TARGET_ADDR** (where to store the plaintext)
STORE an integer into **DMA_BYTECOUNT**
STORE the command **DMA_AES256_DE_INITIAL** into **DMA_Command**
Wait for the task to complete
For the each additional chunk:
STORE an address into **DMA_START_ADDR** (where to find the cipher-text)
STORE an address into **DMA_TARGET_ADDR** (where to store the plaintext)
STORE an integer into **DMA_BYTECOUNT**
STORE the command **DMA_AES256_DE_MIDDLE** into **DMA_Command**
Wait for the task to complete
For the the final chunk:
STORE an address into **DMA_START_ADDR** (where to find the cipher-text)
STORE an address into **DMA_TARGET_ADDR** (where to store the plaintext)
STORE an integer into **DMA_BYTECOUNT**
STORE the command **DMA_AES256_DE_FINAL** into **DMA_Command**
Wait for the task to complete
Retrieve the plaintext from the target area.

Each chunk will be decrypted and stored in the target area before the next command is issued. In some applications, it may be the case that the initial chunk of a message contains a header with a “length” field which indicates how long the message is. After decrypting the first chunk, it may be desirable to use this “length” information to determine exactly how much of the message to decrypt.

Chapter 6: Other Ideas

Quick Summary

• General Thoughts of Other Memory-Mapped I/O Devices

Overview

In this chapter, we outline ideas for I/O peripherals that might be included in a Blitz-64 implementation. The follow devices are suggestive and speculative.

- Lock Controller
- Digital I/O
- SPI / MicroSD Card Slot
- Adjacent Core Links
- HDMI, USB, WiFi, etc.

Lock Controller

In this section, we sketch the design of a device which is novel, hypothetical, unconventional, and speculative.

When scaling systems beyond more than about 16 cores, the traditional approach of implementing shared locks by using atomic operations on shared memory may not work well. The idea here is to off-load the task of synchronization to a dedicated device, in order to improve performance.

This memory-mapped I/O device is used for synchronization between the processors in a multiprocessor system. Consequently, this device will be shared by

all processors in the system. The system may or may not also have shared memory or other shared resources. In systems without shared memory, data might be copied from one private memory to another private memory by a Direct Memory Access (DMA) controller with access to the private memories of several different processors.

A **mutex lock** is normally used to control the access to shared data. Any code which reads and updates shared data is said to be a **critical region**. In many applications, due to the possible unpleasant interaction of concurrent processes, only one thread should be in a critical region at any moment. A mutex lock can be used to enforce this. The lock is either **held** or **free**. Before entering a critical section, every thread must **acquire** the lock, which changes it from “free” to “held” by that thread. After the critical section has been completed, the thread should **release** the lock, which changes it from “held” to “free”.

A lock that is “held” is sometimes said to be “set” or “locked”. A lock that is “free” is sometimes said to be “clear” or “unlocked”.

In a system with only a single core, the implementation of locks is straightforward. Whenever one kernel thread wishes to examine and acquire a lock, it can momentarily disable interrupts (e.g., with the CSRCLR instruction). The thread can check the state of the lock and, if the lock is free, the thread will acquire it before reenabling interrupts. This prevents a thread-switch from occurring while the lock is being manipulated.

In a system with shared memory and multiple cores, disabling interrupts is not sufficient. Other cores, running concurrently, may still interfere with the lock-acquire operation. Another approach is required.

To address this need, most Instruction Set Architectures (ISAs) provide instructions that can be used to implement locking. For example, Blitz includes the CAS (compare-and-set) instruction. The CAS instruction will perform both a read and write to a shared memory location. The instruction will both set the memory location and return the previous value from the read, allowing the software to determine whether the “lock acquire” operation was successful or whether the lock

was already held and the attempt to gain exclusive access has failed. Instructions such as compare-and-set and test-and-set instruction must be executed atomically.[19]

The approach outlined here, which uses a memory-mapped I/O device, is significantly different. It avoids requiring the shared memory to support atomic operations in any way. This would be useful if there is no shared memory. It would also be useful if the system did not support atomic memory operations, perhaps for reasons of efficiency.

Instead of relying on atomic instructions, one can use this I/O device, whose sole purpose is to implement mutex locks. For example, these locks might be used to regulate the access by multiple cores to regions of the shared memory. (For locks used only by a single core, the technique of temporarily disabling interrupts is sufficient, faster, and simpler.)

This memory-mapped I/O device provides 32 special "**lock registers**". Each register is a doubleword (that is, 64 bits wide) and each is addressable as a memory-mapped I/O location, just as any doubleword in memory is addressable. The single page allocated to this I/O device will contain these 32 doublewords, near the beginning of the page, at the offsets shown below.

offset into page (in hex)	
`0000 – 0007`	Lock register #0
`0008 – 000F`	Lock register #1
...	...
`00F8 – 00FF`	Lock register #31

The memory-mapped I/O page for this device contains no other usable locations

Each lock register behaves similarly to any normal doubleword of memory. Each lock register can be read by a LOAD.D instruction. Each lock register can be written by a STORE.D instruction. However, there are differences, which will be described.

[19] There are variations to this approach. For example, the RISC-V approach is called load-reserved/store-conditional (LR/SC), which is also called "load-link/store-conditional (LL/SC). In addition, the RISC-V ISA also includes a number of "atomic memory operations", including instructions such as AMOADD, which will read a value from memory, add a number to it, and then store the result back in memory — all as a single atomic operation.

Each lock register will contain a 64 bit signed value. However, the value 0 has special meaning. A value of zero means that the lock is "free" (i.e., not locked or held by any core). A non-zero value means the lock is held and the value will indicate the identity of the core holding the lock.

A read (e.g., using a LOAD.D) will return the value of the register and work as expected. However, storing into a register (e.g., using a STORE.D instruction) has an unusual behavior. In some cases, the STORE will work; in other cases, the STORE will be ignored and the value will remain unchanged.

To be more specific, any attempt to store a non-zero value into a lock register that previously contained any value other than zero will fail. If the lock register previously contained zero, then the write will succeed and the register will be updated. But if the previous value was nonzero, and an attempt is made to write another non-zero value into the register, then the write will be ignored and the previous value will be unchanged. A write of zero to a lock register will always work.

Another way to think about this is as follows: A lock register works exactly like any other doubleword in memory, except that any attempt to store a non-zero value into a register already containing a non-zero value will be ignored.

The idea is that a single lock register is used to represent and implement a mutex lock. To acquire the lock, a core will write a non-zero value to the register. If the lock was previously free, it will be changed from zero to the number written. The core should follow the STORE.D instruction by executing a LOAD.D to look at the lock's value. If the lock contains the new value, then the lock has been successfully acquired; it it contains any other value, the acquire has failed and must be retried. Later, to release the lock, the core will write a zero into the lock register.

We assume that each core has a been assigned a unique number, which we will call its "**core ID**". We assume that the cores are numbered 1, 2, 3, … N. The idea is that to acquire a lock register, a core will write its number into the lock register using the STORE.D instruction. Then, to determine whether the operation was successful, the core will read the lock register using a LOAD.D. If the number returned is the core's own ID, then the lock was successfully acquired. If the number is anything else, then the acquire operation failed. If the number returned is zero, then it means that the lock was released sometime between the STORE.D and the LOAD.D instructions. Otherwise, the number returned indicates which core holds the lock (or, more correctly, the identity of a core that held the lock at some time in the recent past, and may or may not still hold it).

How can these locks be used. Perhaps each lock register will be used to lock a given region of shared memory for the purpose of enforcing exclusive, sequential access to that memory region. Exactly which critical data is to be protected by each lock is up to the kernel programmer. Perhaps each core will have a region "belonging" to it; each lock will be used to protect the memory associated with a particular core. Or perhaps one lock register will be used as a "master lock" to control access to a shared critical region containing thousands of "secondary" mutex locks. In order to perform an operation on one of the secondary locks, a core would be required to first acquire the master lock.

The lock registers are specified to be 64 bits wide. Recall that aligned LOAD.D and STORE.D instructions are atomic.

A mutex locks is used to regulate and synchronize a set of concurrent "processes". With the Blitz-64 approach, each process must use a different number when executing the STORE.D to an acquire the lock, so that it can determine whether the STORE.D operation successfully acquired the lock. If two different processes are using the same number, there would be no way for one process to tell whether it or the other process successfully acquired the lock.

The lock registers described here are specifically designed to arbitrate between cores, not between threads within a single core. As we said above, each core will use a unique number (e.g., its "core number") in the STORE.D instruction to acquire a lock. This approach uses 64 bit values. With this many bits, we could easily accommodate two fields, one with the core number and the other with a thread ID.

There is nothing particularly special about specifying the number of lock registers to be 32; it was chosen arbitrarily. Perhaps this will change in a future specification. Also note that cores are normally numbered 0, 1, 2, … N-1 but this discussion used 1, 2, 3, … N.

Digital I/O Pins and LEDs

Like many processors, a Blitz-64 chip may contain a number of digital I/O pins. The core will control these pins by accessing a dedicated page.

For example, to change the value of the output pins, software would STORE into specific, predefined locations within this page. To query the current value on the input pins, software would read from locations within this page.

It may be that some locations are reserved for configuring the I/O pins before use. For example, it may be necessary to define whether the pin is "input" or "output" or to send it into the **high Z state**, and this would be done by writing into other specific, predefined locations.

If the implementation includes hardware support for **Pulse Width Modulated** (PWM) signals, then the page would contain additional words which can be used to control the PWM pins.

In some implementations, **analog pins** may be present. If this is the case, then page would contain additional words which can be used to query the analog to digital (ADC) values or to send digital to analog (DAC) values to the output.

In one OS design, user-mode processes would need to invoke **system calls** to access the digital I/O pins. The kernel's syscall handler would access the page directly to perform the operation. In another OS design, the device page is mapped into the address space of a single **pin controller process**. All requests to control and query the digital I/O pins would then go through this pin controller process. In a third OS approach, the kernel will map the relevant page into the address space of any user-mode process wanting to access the digital I/O pins.

Presumably, this device will occupy only a single page. That is, all I/O pins would be controlled from a single page and any process that can access any pins would have access to all pins. But there is plenty of room in the address range for memory-mapped I/O to set aside a unique page for each pin. This would allow fine-grained control over permissions, allow a process to access some pins, but preventing access to other pins.

SPI / MicroSD Card Slot

If interface hardware for microSD card slots is available, then each slot will be assigned to a memory-mapped I/O region.

In initial implementations, it is assumed that the file system and kernel will be located on microSD cards.

Each card slot should be mapped to different pages from the other slots, so that each slot can be individually mapped into the address space of separate controller processes.

Adjacent Core Links

Status: Provisional; details to be determined
Starting Address: 0x4_XXXX_XXXX *< implementation dependent >*
Next Available Address: 0x4_XXXX_XXXX
Size: 0x0_0000_4000 (16 KiBytes)
Number of Pages: 1

Blitz-64 is intended to be used in parallel processing arrays where each node is a Blitz-64 core. Each core may occupy a single chip die (along with its local memory and other components) or there may be multiple cores on each die.

The cores are intended to be arranged in a rectangular array. The array may be 2-dimensional or 3-dimensional, or the cores may be array linearly in a 1-dimensional arrangement.

When arranged in a 3-dimensional array, the directions are called “west”, “east”, “north”, “south”, “up”, and “down”. The size (extent) in each dimension need not be identical. The cores are identified using a coordinate system

In a linearly arranged array, the cores are numbered 0, 1, 2, … M. The western-most core is numbered zero, with the numbers increasing in the eastern direction. You can also think of “left” as corresponding to west and “right” as corresponding to “east”.

In a 2-dimensional array, you can think of the west-east axis as indicating the “column” and the north-south axis as indicating the “row”. The northern-most core is numbered zero, with the numbers increasing in the southern direction.

In a 3-dimensional array, the third axis corresponds to up-down. The uppermost core is numbered zero, with the numbers increasing in the downward direction.

Thus, the cores are numbered from code [0,0,0] in the upper northwestern corner, to core [M-1,N-1,P-1] in the far (lower southeastern) corner where M is the number of columns in the west-east direction, N is the number of rows in the north-south direction, and P is the number of planes in the up-down direction. (Note that this order corresponds to Cartesian coordinates (x, y, z) and not the [row, column] order of matrices.)

Each core can communicate directly with its 6 neighbors. The memory mapped I/O device here is designed to support this communication. The messages can be variable length (up to 1 page in size) and are buffered so that a core need not wait for a transmission to complete. Since each core is expected to have an independent clock, the transmission and corresponding flow control is handled entirely by the hardware. When a transmission is complete, interrupts will be signaled at both ends. The interrupt at the sending end lets the software know that it can initiate a new transmission. The interrupt at the receiving end lets the software know that it can read and process the incoming message.

Each of the six channels will be “full duplex”, which means that the communication in one direction is entirely independent of the communication in the other direction. Communication can occur in both directions simultaneously with no timing interaction or performance impact.

Commentary With 6 communication links, a collection of processor cores may also be arranged in other configurations, such as a 6-dimensional hypercube. A 6-D hypercube arrangement will accommodate 64 cores and the longest path from any core to any other core is only 6.

Contrast this with a 3-dimensional array of 64 processors (i.e., 4 × 4 × 4): the longest path from any core to any other core is 9 (i.e., 3 + 3 + 3).

The difference in path lengths in a hypercube arrangement over a 3-dimensional array becomes more apparent as the dimension and number of cores increases. For 1,024 cores in a 10-D hypercube it is 10::27. The number of wires remains unchanged and is solely determined by the number of cores; each wire has a core at each end, so the number of wires is cores × links ÷ 2. But wiring in our 3-D universe becomes messy.

The real world is 3 dimensional (although physicists might correct me) and many computations are tied to this dimensionality, so practical applications tend to map naturally onto 3-D processor arrays.

Note In an earlier discussion concerning the Lock Controller device, we discussed how each core could "acquire" a lock by writing its core ID number into special location in the memory-mapped I/O device that is dedicated to controlling locks. The Lock Controller uses a value of zero to indicate that a lock is "free", so the numbering of the core ID values must begin with 1.

Here, in the discussion of arrays of cores, we are numbering cores so that the home / master core at the origin of a 3-D array is given the address [0,0,0]. If the array is only 1-D and the cores are laid out in a line, they are given addresses [0], [1], … [M]. In other words, the cores are numbered:

Array Address	Core ID for Locking
[0]	1
[1]	2
[2]	3
…	…

These two numbering systems are different; be careful of confusing them.

HDMI, USB, WiFi, etc.

If interface hardware for other devices is available, then each device will be assigned to a memory-mapped I/O region.

Chapter 7: The CONTROL and CONTROLU Instructions

Quick Summary

• The CONTROL and CONTROLU instructions are implementation-dependent
  — Used by the Verilog/FPGA implementation
• These instructions are used to…
  — Control the digital I/O pins
  — Control the UART device
  — Debug the TLB registers
  — Debug the Verilog/FPGA core

CONTROL and CONTROLU

CONTROL and CONTROLU are two machine instructions which have the same syntax. They take a

- Destination register
- Source register
- 16 bit immediate value

Here are two examples:

```
        control     r3,r6,1234
        controlu    r7,r0,0x56ab
```

The definition and specification of these instructions is entirely implementation dependent. Below, we will describe how they are implemented in the emulator and in the MicroBlitz Verilog/FPGA implementation.

In the Emulator and Verilog/FPGA

The CONTROL and CONTROLU are implemented identically in both

- The Blitz emulator
- The MicroBlitz Verilog/FPGA implementation

In both implementations, only CONTROLU is implemented and used. The CONTROL instruction is left unimplemented, as a stub instruction. The CONTROL will not be discussed further.

The CONTROLU instruction, as described here, was included to assist in the development of the Verilog/FPGA implementation. Initially, the Verilog/FPGA did not support memory-mapped I/O. But since human-readable text output was needed and very useful, the behavior described below was added to CONTROLU.

The immediate value is used to select an operation:

value	***operation***
0	DIGITAL_READ
1	DIGITAL_WRITE
2	HALT
3	SERIAL_STAT
4	SERIAL_RECV
5	SERIAL_SEND
6	ENABLE_KERNEL
7	SET_STATUS
8	TLB_DEBUG

For convenience, assembler programs contain the following definitions:

```
DIGITAL_READ:   .equu   0
DIGITAL_WRITE:  .equ    1
HALT:           .equ    2
SERIAL_STAT:    .equ    3
SERIAL_RECV:    .equ    4
SERIAL_SEND:    .equ    5
ENABLE_KERNEL:  .equ    6
SET_STATUS:     .equ    7
TLB_DEBUG:      .equ    8
```

In the emulator, other values will interrupt program execution with a message. In the Verilog/FPGA implementation, other values will cause an Illegal Instruction Exception.

Digital Read / Write

```
controlu   XXX,r0,DIGITAL_READ
controlu   r0,XXX,DIGITAL_WRITE
```

In the DIGITAL_READ operation, the source register is ignored and the destination register is modified.[20] In the Verilog/FPGA implementation, the digital input comes from the 8 of the slide switches on the FPGA board and the corresponding binary value is placed in bits [7:0], with the upper bits set to zero. When being emulated, the emulator will momentarily halt execution and prompt the user to input a value.

In either case, a 64-bit binary value will stored in the destination register.

In the DIGITAL_WRITE operation, the destination register ignored and not modified. Instead, a 64 bit value comes from the source register. In the Verilog/FPGA implementation, the lower order 16 bits are sent to the 4 digit seven-segment and displayed as four hex digits. The upper bits are ignored. When being emulated, the emulator will momentarily halt execution and display the value.

---

[20] When registers are ignored and not needed, register r0 can be specified. Here, “xxx” symbolizes a source or destination register that is needed and relevant.

Halt

```
controlu   r0,r0,HALT
```

This instruction will stop execution. In the Verilog/FPGA implementation, this instruction will send the core into the "sleep" state, where it will remain until a reset occurs. In the emulator, either the emulator itself is terminated (if the auto-go option "-g" was present) or the user is popped up into the debugger (otherwise).

Both the source and destination registers are ignored.

Serial / UART Control

```
controlu   XXX,r0,SERIAL_STAT
controlu   XXX,r0,SERIAL_RECV
controlu   r0,XXX,SERIAL_SEND
```

With SERIAL_STAT, the destination register will be loaded with a 64 bit value, with the following bits:

[63:2]	Unused; always returned as zero
[1]	Output ready
[0]	Input available

With SERIAL_RECV, the input character will be moved in the lower 8 bits of the destination register and the upper bits are set to zero.

With the SERIAL_SEND, the byte in the source register will be sent to the output, with the upper 56 bits of the register being ignored.

With the emulator, the serial channel is always ready, so the SERIAL_STAT will always have the lower two bits set to 1. SERIAL_RECV will simply "get" the next byte and SERIAL_SEND will "put" a byte to the output without any delay as measured by the cycle count.

With the Verilog/FPGA implementation, there will be a delay. The code should always check the status before using SERIAL_RECV or SERIAL_SEND. Attempting to

send or receive before the system is ready results in incorrect data values being transferred.

ENABLE_KERNEL

```
controlu   XXX,r0,ENABLE_KERNEL  # grab csr_status; then set mode
```

This operation loads 0x0000_0000_0000_0001 into CSR_STATUS, which will disable interrupts and place the core in Kernel Mode. It also moves either 0 or 1 into the destination register to indicate whether the core was previously in Kernel Mode (1=Kernel Mode, 0=User Mode).

SET_STATUS

```
controlu   r0,XXX,SET_STATUS     # set csr_status from reg.
```

This operation moves the value in the source register into CSR_STATUS.

TLB_DEBUG

```
controlu   regD,Reg1,TLB_DEBUG   # copy TLB[r1] to RegD
```

With this operation, the value in the source register is used as the number of one of the TLB registers. This operation moves the value of the selected TLB register into the destination register.

This operation is useful and important for verifying the proper functioning of the Memory Management Unit (MMU) since the TLB registers are not otherwise directly accessible.

The TLB registers in the Verilog/FPGA implementation are 72 bits long. In both the emulator and the Verilog/FPGA implementations, the TLB register is reduced[21] to a 64 bit value with the following format, which is moved into the destination register.

Bits	Size	
63 … 48	16	Address Space Identifier (ASID)
47 … 27	21	Virtual Page Number
26 … 5	22	Physical Page Number
4	1	C: Copy Bit
3	1	D: Dirty Bit
2	1	W: Writable Bit
1	1	X: Executable Bit
0	1	V: Valid Bit

[21] In Page Table Entries (PTEs) and in the TLB registers, the Physical Page Number is 30 bits. The Physical Page Number is reduced from 30 bits to 22 bits by dropping the most significant 8 bits. Note that 72-8 = 64.